Due date: Thursday February 28 11:59:59pm EST.

Getting started: To create your initial Lab 3 repository, please visit the repository creation page at https://6004.mit.edu/web/spring19/user/labs/lab3. Once your Lab 3 repository is created, you can clone it into your VM by running:

```
$ git clone git@github.mit.edu:6004-spring19/labs-lab3-{YourMITUsername}.git lab3
```

Turning in the lab: To turn in this lab, commit and push the changes you made to your git repository. After pushing, check the course website (https://6004.mit.edu, Labs/Didit tab) to verify that your submission was correctly pushed and passes all the tests. If you finish the lab in time but forget to push, you will incur the standard penalties for late submissions.

Check-off meeting: After turning in this lab, you are required to go to the lab for a check-off meeting. See the course website for lab hours.

1 Introduction

The combinational logic in our computers plays a key role in forming many key computation structures including adders and subtractors, multiplexers, decoders, and an arithmetic logic unit (ALU) which we learned is a fundamental building block of the CPU. Combinational logic takes multiple boolean inputs and produces multiple boolean outputs where the outputs are a function of the inputs at a specific time only. In other words, combinational logic handles purely current combination of inputs and has no notion of previous state or memory.

In this lab, you will get a chance to build various combinational circuits which consist of basic logic gates (e.g. and, or, xor, and etc.) and perform useful tasks in Bluespec. Bluespec is a high-level hardware description programming language. What this means is that it is specialized to design, simulate, and implement digital “hardware” systems. Therefore, a hardware description language like Bluespec is synthesized into actual digital circuits and capable of accurately describing very complex digital hardware system whereas popular general-purpose software programming languages like C/C++ and Python translated into a sequential series of assembly instructions cannot quite describe massively parallel computer system hardware.

With that in mind, you will write fully synthesizable Bluespec programs that describe useful digital systems. In order to make sure you have a good grasp of Bluespec basic syntax and writing functions by completing the lab, we put restrictions on what subset of Bluespec you can use in your designs. Although Bluespec has and ($\land$), or ($\lor$), xor ($\oplus$), and not ($\sim$) operators and higher-level constructs (such as if statements, for loops, +, -, and ==), you are not allowed to use them in this lab. Using any of these will cause a compiler error. In the error, it should tell you in all caps what symbol you used.

For this lab, unless otherwise stated, you are only allowed to use constants, the base functions shown below (which are defined in Common.bsv), and any functions you define yourself using these functions. Finally, you should NOT modify Common.bsv since it will be overwritten by the auto-grading script.

To pass the lab you must complete and PASS all of the exercises.

```plaintext
function Bit#(1) and1(Bit#(1) a, Bit#(1) b);
function Bit#(1) or1(Bit#(1) a, Bit#(1) b);
function Bit#(1) xor1(Bit#(1) a, Bit#(1) b);
function Bit#(1) not1(Bit#(1) a);
function Bit#(1) multiplexer1(Bit#(1) sel, Bit#(1) a, Bit#(1) b);
```
2 Multiplexer a.k.a “Data Selector”

A multiplexer, also called a data selector, is a combinational logic circuit designed to choose a single output signal from multiple input signals based on select signals. Let’s first take a look at what a multiplexer looks like in basic logic gates. We saw in lecture that a 2-way multiplexer can be built using 1 Inverter, 2 AND gates, and 1 OR gate as shown in Figure 2. In boolean algebra representation, \( \text{out} = a\overline{s} + b s \) such that an output is \( b \) if a select signal \( s \) is 1 and \( a \) if \( s \) is 0. In Bluespec, a 2-way multiplexer that takes two 1-bit inputs and produces a single 1-bit output can be implemented as follows using the given base functions.

Figure 2: 2-way multiplexer implementation using basic gates.

```bluespec
// A 2-way multiplexer in Bluespec which takes two 1-bit inputs
// and produces a single 1-bit output
function Bit#(1) multiplexer1(Bit#(1) sel, Bit#(1) a, Bit#(1) b);
    return or1(and1(a, not1(sel)), and1(b, sel));
endfunction
```

Alternatively, we can also implement it using a Bluespec conditional statement as shown below. Note that for this lab, you are not allowed to use conditional statements unless otherwise specified.

```bluespec
// A 2-way multiplexer in Bluespec which takes two 1-bit inputs
// and produces a single 1-bit output
function Bit#(1) multiplexer1(Bit#(1) sel, Bit#(1) a, Bit#(1) b);
    return (sel == 0)? a: b;
endfunction
```

# Python equivalent code

```python
def multiplexer1(sel, a, b)
    return a if sel == 0 else b
```

You will often have to deal with inputs whose sizes are more than 1-bit long and consequently being
able to utilize a simple function with small-sized inputs as a building block to perform larger and more complex tasks will be very useful. A simple example of this includes expanding a 1-bit multiplexer into a 2-bit multiplexer as follows.

```plaintext
// A 2-way multiplexer which takes two 2-bit inputs and produces a single 2-bit output
function Bit#(2) multiplexer2(Bit#(1) sel, Bit#(2) a, Bit#(2) b);
return {multiplexer1(sel,a[1],b[1]), multiplexer1(sel,a[0],b[0])};
endfunction
```

First of all, function `Bit#(2)` tells us that this function returns a 2-bit output. The function parameters in `multiplexer2(Bit#(1) sel, Bit#(2) a, Bit#(2) b)` tell us that the function takes a 1-bit select value, `sel`, and two 2-bit inputs `a` and `b`. Note that bit concatenation is used in the above example to return a 2-bit output signal by concatenating two 1-bit multiplexers using `{ , }`. As a result, a 2-bit multiplexer takes two 2-bit inputs `a` and `b` and produces a 2-bit output based on a 1-bit selection value `sel`. For those who are not familiar with the notion of bit concatenation, we provide a simple example of how this works in Bluespec.

Assume that you have two binary numbers `a` and `b` as shown below whose sizes are 3-bits and 7-bits, respectively. You would like to concatenate these two numbers.

```plaintext
Bit#(3) a = 3'b010; // a is 3-bit long
Bit#(7) b = 7'b1011110; // b is 7-bit long
```

Using `{ , }` operator, you can concatenate these two number and produce a 10-bit output as follows.

```plaintext
Bit#(10) abconcat = {a,b}; // As a result, abconcat = 10'b0101011110
```

Finally, you can access a specified range of any binary number using bit selection as follows.

```plaintext
Bit#(3) bselect = b[2:0]; // bselect = 3'b110
Bit#(4) bselect = b[6:3]; // bselect = 4'b1011
```

### Building and Testing Your Circuits

You will build your code with `make`. If you just type `make` it will build all of the exercises. You can instead pass in a target so that only one of the exercises will be built like

```
make <target>
```

This will then create a program `<target>` that you can run which simulates the circuit. It will run through a set of test cases printing out each input and whether or not it fails to match the expected output. If it passes all the tests it will print out PASSED at the end.

- To build all the targets, run
  ```
  make all
  ```
- To build and test everything, run
  ```
  make test
  ```

### 3 Bit Scan Reverse

`bit_scan_reverse` determines the index of the first non-zero bit scanned from the largest index. This is equivalent to taking the log₂ of the input and rounding it to the next smaller integer (i.e. ⌊log₂(Input)⌋). Note that `bit_scan_reverse(0)` is undefined, so we will not check your output in this case. For example,

```plaintext
bit_scan_reverse(4'b1000) = 3
bit_scan_reverse(4'b0110) = 2
bit_scan_reverse(4'b0001) = 0
```
Exercise 1 (10 points): In CombDigitalSystems.bsv, implement the `bit_scan_reverse` function for a 4-bit input using only the basic gates provided in Common.bsv. Figure 3 shows the skeleton code for this function. 

Hint: We recommend writing out a truth table for the Bit-scan Reverse function and trying to deduce how the 4-bit input value relates to the 2-bit output. Then, try drawing the circuit with logic gates before writing any Bluespec code.

You can build and test your circuit by running:

```
make bit_scan_reverse_test
./bit_scan_reverse_test
```

4 Power of 2

The `is_power_of_2` circuit determines whether its input is a power of 2. A binary number is a power of 2 if it has exactly a single 1.

Exercise 2 (10 points): In CombDigitalSystems.bsv, implement `is_power_of_2` for a 4-bit input, which returns 1 if an input is a power of 2 and 0 otherwise.

You can build and test your circuit by running:

```
make is_power_of_2_test
./is_power_of_2_test
```

5 log₂ of Powers of 2

We now want to combine the previous two circuits. We want a circuit `log_of_power_of_2` that will output 0 if the input is not a power of 2, and `log₂` of the input if it is a power of 2.

Exercise 3 (15 points): In CombDigitalSystems.bsv, implement `log_of_power_of_2` for a 4-bit input using your previous two circuits as well as any of the basic gates. 

Hint: Think about how you can simplify the implementation by importing `bit_scan_reverse` and `is_power_of_2` functions and expanding the given 1-bit multiplexer function to a 2-bit multiplexer.

You can build and test your circuit by running:

```
make log_of_power_of_2_test
./log_of_power_of_2_test
```

6 Equality Testing

We often need to know whether two numbers are the same. Write a function `equal` that returns 1 if two 8-bit numbers are the same, and 0 otherwise.
Exercise 4 (10 points): Implement the equal function in CombDigitalSystems.bsv using only the basic gates.

You can build and test your circuit by running:

```
make equal_test
./equal_test
```

7 Vector Equality Testing

We are often interested in comparing groups on values for equality. For example, given two vectors of four 8-bit (1-byte) values, we would like to produce a circuit that returns a 4-bit result, where the $i^{th}$ bit is 1 if the $i^{th}$ elements of the vectors are equal, and 0 otherwise. Design a vector_equal circuit that performs this function. This circuit should take two 32-bit (4-byte) inputs, with each byte representing a different element of the 4-element vector. For example,

- `vector_equal(0xaabbccdd, 0xaa11ccdd) = 0b1011`
- `vector_equal(0xaabbccdd, 0x0011ccdd) = 0b0011`
- `vector_equal(0xaabbcc00, 0x0011ccdd) = 0b0010`

Exercise 5 (10 points): Implement the vector_equal function using your equal circuit as well as the basic gates in CombDigitalSystems.bsv.

You can build and test your circuit by running:

```
make vector_equal_test
./vector_equal_test
```

8 Seven-Segment Decoder

A seven-segment display shows numbers by lighting up a subset of the 7 LED segments. Each segment is traditionally lettered from a to g as shown in Figure 4, and each segment has its own input signal for enabling and disabling the segment. Figure 5 shows how the numbers should be displayed on the seven-segment display.

![Seven-segment display](image)

Figure 4: Seven-segment display.

You are in charge of designing a digital logic controller to take in a binary number between 0 and 9 inclusive, and output the 7-bit control signals to drive a seven-segment display. The output control signal takes the form $abcdefg$ where $a$ is the most significant bit of the output (bit 6) and $g$ is the least significant bit of the output (bit 0). When the input signal is outside the range of legal inputs, you should make the display show an E indicating an error.
Exercise 6 (15 points): Implement the `seven_segment_decoder` function in `CombDigitalSystems.bsv`. For this exercise only, in addition to the basic gates in `Common.bsv` you can also use `case` statements in your Bluespec implementation.

Note that `case` statements are equivalent to a series of nested if-then-else statements. As a simple example, the case and nested if-then-else statements shown below are equivalent.

```verbatim
function Bit#(7) seven_segment_decoder(Bit#(4) input_binary_number);
    Bit#(7) ret = 7'b1001111; // value for "E"
    return ret;
endfunction

Figure 6: Skeleton code for `seven_segment_decoder`.

Exercise 7 (15 points): Implement the `population_count` function in `CombDigitalSystems.bsv` using only the basic gates. Hint: You may want to use a full adder function to count the number of 1’s in the input. An example 1-bit full adder and its truth table are shown in Figure 7.

You can build and test your circuit by running:

```
make seven_segment_test
./seven_segment_test
```

9 Population Count

Next, you are in charge of implementing a population count circuit, which counts the number of 1’s in the input. Your population count circuit should take in a 4-bit input and return an output representing the number of 1’s in the input.

Exercise 7 (15 points): Implement the `population_count` function in `CombDigitalSystems.bsv` using only the basic gates. Hint: You may want to use a full adder function to count the number of 1’s in the input. An example 1-bit full adder and its truth table are shown in Figure 7.

You can build and test your circuit by running:

```
make seven_segment_test
./seven_segment_test
```
make population_count_test
./population_count_test

10 Comparator

As your final task, you are in charge of implementing a 4-bit comparator. Write a function is_geq that takes in two 4-bit binary unsigned numbers a and b and returns 1 if a is greater than or equal to b, and 0 otherwise.

Exercise 8 (15 points): Implement the is_geq function in CombDigitalSystems.bsv using only the basic gates.

Hint: You may want to implement a function that performs a bit-wise comparison and then extend it to perform a 4-bit comparison.

You can build and test your circuit by running:

make is_geq_test
./is_geq_test

11 Circuit Performance Analysis

After building the digital system, it is crucial to analyze performance of your system, look at implementation trade-offs, and further improve the system. In order to do this, you are provided with a synthesis tool that translates your Bluespec designs into optimized gate-level implementations using only basic logic gates such as Inverter, NAND, NOR, etc.

We will use synth, a simple synthesis tool. Synthesis tools require three types of input: the circuit to be synthesized, the standard cell library of gates to use to implement the circuit, and the optimization objective (area, delay, power, or some combination of them). synth comes with several standard cell libraries, all of them derived from the open-source FreePDK45 library (https://research.ece.ncsu.edu/eda/freepdk/freepdk45/). This is representative of 45 nm silicon fabrication technology introduced in 2008. By default, synth optimizes for delay, but this is controllable by specifying a target delay. synth will try to reduce the circuit’s area as long as its delay is under the target. Under the hood, synth uses the Bluespec compiler and the Yosys open synthesis suite (http://www.clifford.at/yosys/).

In this part of the lab, we will first learn how to use synth by synthesizing and analyzing several circuits. These exercises will build your intuition of delay and area implementation trade-offs and of the optimizations synth can perform which will be very useful when it comes to designing and building much more complex digital systems. Using synth, we are going to analyze two systems you already built in this lab: bit_scan_reverse and seven_segment_decoder.

First, your bit_scan_reverse function can be synthesized by running:
synth CombDigitalSystems.bsv bit_scan_reverse

synth reports three pieces of information. First, it reports three summary statistics: the number of gates, the total area these gates take (in square micrometers), and the circuit’s critical-path delay (i.e., the longest propagation time between any input-output pair) in picoseconds. Second, it reports the delay across the different gates of the critical path. Third, it shows a breakdown of the types of basic gates utilized in your program and their area. synth can also produce circuit diagrams which allows you to see the circuit implementation of your program. To do that, run:

```bash
synth CombDigitalSystems.bsv bit_scan_reverse -v
```

This will produce a diagram in `bit_scan_reverse.svg`. You can open it by running `firefox bit_scan_reverse.svg` (or use another SVG viewer).

**Discussion Question 1** Does the circuit diagram shown in `bit_scan_reverse.svg` match your Bluespec implementation of `bit_scan_reverse`? Prove that is the case by using boolean algebra.

By default, synth uses the basic standard cell library, which only has a buffer (the identity function: 0 returns 0, and 1 returns 1), an inverter, and two-input NAND and NOR gates. You can control the library used with the `-l` flag. Let’s try using the extended library, which also has AND, OR, XOR, and XNOR gates, as well as 2, 3, and 4-input gates. To see the difference, we will analyze a more complex digital system, `seven_segment_decoder` for which is not intuitive for us to count the number of basic gates. To do this, run:

```bash
synth CombDigitalSystems.bsv seven_segment_decoder -l extended
```

**Discussion Question 2** How does `seven_segment_decoder` change when synthesized with the extended library? What gates are used now vs. with the basic library? How does this affect area and delay? You can also visualize the new circuit by running

```bash
synth CombDigitalSystems.bsv seven_segment_decoder -l extended -v
firefox seven_segment_decoder.svg
```

By default, synth tries to minimize delay by setting a target delay of 1 ps, which is obviously unachievable in this technology. You can control the target delay with the `-d` flag. A relaxed delay requirement will cause the synthesis tool to optimize for area instead. This way, you can trade off delay for area (in current technology power is also a crucial consideration, even more so than area; but power analysis is a complex topic, so in this course we will focus on area and delay). For example, the following command uses a target delay of 1000 ps:

```bash
synth CombDigitalSystems.bsv seven_segment_decoder -l extended -d 1000
```

**Discussion Question 3** Synthesize `seven_segment_decoder` using the command above. How do its area and delay change vs. the previous (delay-optimized) circuit?

**Discussion Question 4** When it comes to designing digital system, we would like to reduce area and power and improve speed of the system. Using synth tool, report the number of basic gates and area in each problem and see if you can further optimize your designs. During a check-off, you should be able to justify your designs in terms of the number of utilized basic gates and tell us how you went about further optimizing your designs.